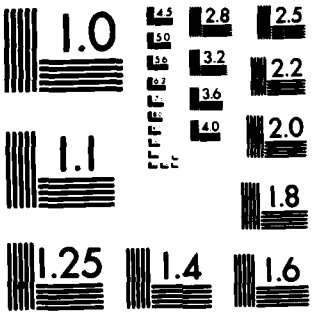
AD-A170 920 REWRITE RULE MACHINE(U) SRI INTERNATIONAL MENLO PARK CA 1/1
J GOGUEN ET AL. JUL 86 N00014-85-C-0417

UNCLASSIFIED F/G 9/2 NL

END
DATE FILMED
9-86



SRI International

Progress Report on the Rewrite Rule Machine

July 1986

By: Joseph Goguen, Program Manager
Claude Kirchner, International Fellow
Sany Leinwand, Senior Research Engineer
José Meseguer, Senior Computer Scientist
Timothy Winkler, Computer Scientist

SRI Project ECU 1243

Prepared for:

Office of Naval Research
Information Sciences Division
800 N. Quincy St.
Arlington, VA 22217-5000

Attn: Dr. Charles Holland, Code 1133

Contract No. N00014-85-C-0417

SRI International
333 Ravenswood Ave.
Menlo Park, CA 94025
(415) 326-6200
TWX: 9100-373-2046
Telex: 334486

SRI International®
333 Ravenswood Ave • Menlo Park, CA 94025
415 326-6200 • TWX 910-373-2046 • Telex 334-486

Progress Report on the Rewrite Rule Machine

J. Goguen, C. Kirchner, S. Leinwand, J. Meseguer, T. Winkler
SRI International, Computer Science Lab

July 15, 1986

Abstract: Perhaps the most significant accomplishment of the Rewrite Rule Machine (RRM) project so far has been to explore the hardware and software implications of a novel model of computation, concurrent tree rewriting. This model serves as a bridge between easily programmed Ultra High Level Languages (UHLLs), featuring implicit concurrency, and advanced architectural designs having unprecedented performance (thousands of MIPS). Additional accomplishments include: (1) construction of a (high level) instrumented simulator for a declarative UHLL (called OBJ) running on the RRM; (2) demonstration that OBJ can be used effectively to program the RRM; (3) demonstration that large amounts of concurrency are available in typical OBJ programs; (4) design of even more powerful UHLLs for the RRM that combine object-oriented, functional and logic programming; (5) exploration of more detailed computational models and hardware designs for the RRM; and (6) progress toward a powerful graphical notation for UHLL programming. We find these results very encouraging and look forward to their fruition in a prototype machine.

1 Introduction

This report summarizes some background and recent progress of the Rewrite Rule Machine (RRM) project at SRI International. The sections following this introduction discuss our model of computation, programming languages, simulation results, and architectural designs.

1.1 Goals of the Project

The purpose of this project is to design and test a prototype high speed, parallel computer architecture for executing ultra high-level languages (UHLLs) to support development of extremely large software systems with stringent correctness requirements. Such UHLLs can be used to write extremely powerful environments including "intelligent" editors, compilers, libraries of reusable software, rapid prototyping tools, formal specification languages, program verifiers, debuggers, test case generators, etc. The language supported by the environment could be a conventional programming language (e.g., Ada), even though the system itself is written in an UHLL; similarly, the target machine for the environment could be a conventional machine, even though the environment runs on the RRM.

The RRM will consist of a large number of loosely connected processors, each with custom VLSI to process tree-structured data independently and very efficiently. The UHLLs considered will combine the paradigms of object-oriented, functional, and logic programming, plus other powerful features, including parameterized programming, graphical programming, sophisticated error handling, and powerful type systems. Compilers will convert UHLL programs into sets of rewrite rules for execution on the RRM.

This effort is aimed at significantly advancing the state of the art in software development, so as to make feasible enormous software development efforts like that required by the Strategic Defense Initiative (estimated at 10 to 35 million lines of highly reliable code). We believe that such an advance can only be achieved by radically new approaches in both hardware and software. The RRM is, however, a general purpose large scale parallel machine that is well-adapted to any kind of symbolic computation; for example, it should excel at artificial intelligence applications such as high-level vision, expert systems, and natural language processing. Hardware simulation is another promising application area.

1.2 Key Issues

We feel that *programmability* is one of the most critical issues blocking further progress in parallel computation: it does little good (for most problems) to provide lots of processors if the programmer has to explicitly assign processes to processors. We feel that the best approach to combining hardware efficiency with programming ease and flexibility is to have a model of computation that provides a simple bridge between a powerful UHLL and the hardware itself. We argue that *tree rewriting* is such a bridge. On the one hand, recent work on programming language semantics shows how to implement advanced languages with tree rewriting; on the other hand, the RRM will process tree-structured data very efficiently. This is a radically different computational model from that of von Neumann machines, and also differs from other new machine projects in important ways. We have taken OBJ2 [1] as our basis. This is a very advanced functional UHLL with a uniquely powerful generic module facility and type system. This basis has been extended to include logic programming [2] and object-oriented programming [3].

From the software point of view, tree processing means that manipulations can easily be described in a way that is independent of the order of execution, and that also provides ample opportunities for concurrent execution. The basic mode of tree processing is called *tree* (or term) *rewriting* (or replacement or reduction), and refers to the replacement of one subtree by another, whenever a tree-structured template is matched. A *rewrite rule* consists of two such templates, one for the subtree to be replaced, and another that determines what it is to be replaced by. See Section 2 for further discussion.

The feasibility of writing nontrivial programs with rewrite rules has been shown by experience with programming languages like OBJ, Hope, Miranda, FP and work of Hoffman and O'Donnell; for example, several different language interpreters have been written in OBJ, including one for OBJ.

Some ideas that are basic to our design include the following:

- Tree structure supports concurrent processing in a natural way;
- Interprocessor communication required during computation is largely local;
- Each processor needs to know only a small percentage of the rules; and
- Each processor is especially designed for tree processing.

1.3 Summary of Progress

The Rewrite Rule Machine project blends state-of-the-art research in hardware and software. Because of the concurrency inherent in our concurrent tree rewriting model of computation, we expect unprecedented performance (many thousands of MIPS). And because of the closeness of the model to declarative programming, we also expect unprecented ease of programming in Ultra High Level Languages, further augmented by a powerful graphical notation. The following are some of the main achievements of the project:

1. Exploration of the hardware and software implications of a novel and promising model of computation, *concurrent term rewriting*.

2. Instrumented simulator for OBJ using the Concurrent Tree Rewriting computational model.

3. Demonstration that OBJ can be used very effectively for programming the RRM.

4. Demonstration that there is enormous concurrency in OBJ programs.

5. Improvement on the OBJ interpreter.

6. Development of multiparadigm UHLLs for the RRM.

7. Progress in architectural design.

8. Installation of Symbolics machine and porting OBJ2 to it; installation of Sun-3 network.

9. Progress on the development of a powerful graphical notation for writing, reading, and modifying UHLL programs.

2 Models of Computation

At the highest level of abstraction, RRM computation can be seen as rewriting a tree at multiple sites concurrently. Less abstractly, such a (possibly very large) tree can be partitioned into pieces that are assigned to different processors, with each processor doing concurrent rewriting on its own fragment of the tree; this gives a second level of computational modelling, which we call *partitioned concurrent rewriting*. Considering how trees are represented and transformed inside processors gives an even more concrete, third level of description, which is addressed in Section 5.

2.1 Tree-structured Data and Computation

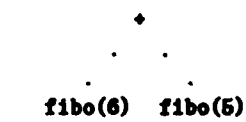
There are many applications in which the data are naturally tree-structured (in the sense that there is a natural hierarchy, with a "root" node at the top and with one or more branches from each node that is not a tip node), and for which tree rewriting is a highly efficient and natural mode of execution. Some examples of naturally occuring tree structures are:

- Menus, such as occur in interactive graphics.
- Expressions, such as $(A + B)(A^2 + 3)$ and, more generally, programs.
- Natural language syntax, and many other structures that occur in natural or artificial languages, such as plans and explanations.

In fact, tree structure is a fully general form of data structuring, since any computable function can be seen as a computation on tree-structured data that rewrites subtrees into other subtrees. In particular, such processes as selecting a particular item from a menu, evaluating an arithmetic expression, verifying a program, assigning a meaning to a sentence, editing a program, constructing a plan, and compiling or interpreting a program, can all be conveniently described as tree rewriting processes. Moreover, tree structure helps programmers and users visualize what programs are doing, and is thus very convenient for the graphical user interface that we are contemplating for languages to run on the RRM.

2.2 Concurrent Tree Rewriting

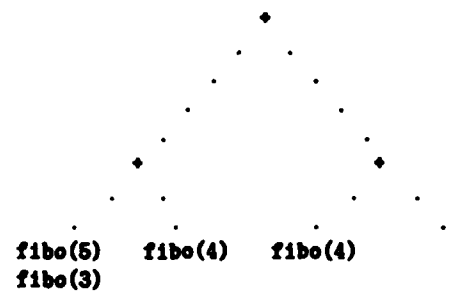
The RRM computation is concurrent tree rewriting. Data are trees (or terms) with internal nodes labelled by function symbols and leafs labelled by constants. Programs are sets of equations, which are interpreted as rewrite rules. Then computation proceeds by *subtree replacement*, which consists of matching the lefthand side of a rewrite rule with a subtree, and then replacing that subtree with the corresponding instance of the righthand side of the rule. For example, one step of evaluating `fibo(6) + fibo(5)` (where `fibo` is the function symbol for Fibonacci numbers), as represented by the tree



matches the rewrite rule

(*) `fibo(N) = fibo(N - 1) + fibo(N - 2)`

to `fibo(6)`, yielding `(fibo(6-1) + fibo(6-2)) + fibo(5)`. If the rewrite rule (*) had been applied to `fibo(5)` instead, one step of rewriting would have yielded `fibo(6) + (fibo(5-1) + fibo(5-2))`. Notice that the rule (*) actually applies concurrently to both `fibo(6)` and `fibo(5)`, yielding `(fibo(6-1) + fibo(6-2)) + (fibo(5-1) + fibo(5-2))`. In addition to applying the same rule in parallel to several subtrees, one can also apply several different rules concurrently. For example, applying a rule for subtraction simultaneously with the rule (*) would transform the original tree into the tree



in one step of concurrent rewriting. This process continues until there are no more matches, when the expression is said to be *reduced*. This simple example shows that tree rewriting is by its very nature concurrent. We emphasize that the concurrency is implicit in the rewrite rules themselves, and *no explicit concurrency constructs are required in the language*; we see this as a major advantage.

2.2.1 Locality of Communication

An important property of tree rewriting is that usually only *local* information is needed to determine whether or not a lefthand side matches, and then if it does, to carry out the corresponding subtree replacement[1]; that is, only a small number of neighboring cells need to be involved. This implies that only local communication is needed for tree rewriting, and makes it possible to use a relatively inexpensive communication network.

[1]This holds for the "left linear" case where there are no repeated variables in the pattern to be matched; otherwise, it may be necessary to check equality of arbitrary subterms. However, left linear rules are much more common than non-left linear rules, and in practice, one can convert programs into non-left linear form.

2.2.2 Theoretical Study of Concurrent Tree Rewriting

We have begun the theoretical study of our most abstract computational model, concurrent tree rewriting. Although there is an extensive literature on sequential tree rewriting, which has been extremely useful in our research, concurrent tree rewriting is a new concept. The first thing to investigate is to what extent standard results extend from the sequential case to the concurrent case. Two of the most basic properties of tree rewriting systems 'lift' from sequential rewriting: if the system is (sequentially) terminating (there are no infinite sequences of rewritings) or Church-Rosser (which implies that reduced forms are unique if they exist) then so is concurrent rewriting. However, some other conditions, such as "weak termination" (that each tree has at least one reduced form) do not extend from sequential rewriting to concurrent rewriting. More restricted tree rewriting systems, such as left linear systems (see footnote 1) and non-overlapping systems, may overcome these difficulties.

2.2.3 Tree and Rule Set Partitioning

When the tree to be reduced is large, it must be partitioned into subtrees residing on different processors, each concurrently reducing its own subtree. This will require communication when a rule match overlaps a partition boundary.

In order to more efficiently use a network of processors each having small local memory, we partition the set of rules, attempting to assign to a given processor only the rules it will need. This can be done in several complementary ways, of which the main ones are:

- By module: The hierarchy of OBJ modules indicates some important relations of grouping and dependency among rules.
- By complexity of lefthand sides of rules:
  1. Left linear non-overlapping rule sets can be compiled into very efficient matching algorithms [4][2].
  2. Non-left linear non-overlapping rule sets will be more costly to match.
  3. Self-overlapping rules (their lefthand side overlaps with itself) can be grouped into a single "try last" rule subset; among these, non-left linear rules constitute the worst case.

  Rules that might overlap should be delayed as long as possible, since communication is needed to prevent meaningless results. This suggests partitioning the rule set into non-overlapping rule subsets.
- By flow analysis: rules can be clustered into subsets such that, given a subtree to which a given rule applies, other rules in the same cluster are more likely to apply to the subtree than are rules outside the cluster.

2.2.4 Concurrent Rewriting Strategies

The notion of *E-strategy* (*E* is for *evaluation*) was introduced in sequential OBJ2 [1] as a powerful and flexible way to control the order of evaluation, so as to improve efficiency of execution. For example, given a 3 argument operator f with strategy [1 2 0 3] and an expression $f(t_1, t_2, t_3)$ to be reduced, the first argument t_1 (indicated by the number 1) must be reduced first, say to t_1', before reducing the second argument t_2 to t_2' (indicated by the number 2); then we must attempt to rewrite at the top (indicated by the number 0) of $f(t_1', t_2', t_3)$ before finally going on to reduce t_3. This kind of sequential evaluation is not appropriate for concurrent rewriting; but fortunately, the interpretation of E-strategies can be extended. For the example given above, we would begin by evaluating all three major

[2]Some additional work will be needed, however, to understand the application of [4] to the concurrent case.

subtrees of $f(t_1, t_2, t_3)$ concurrently, until its first and second subtrees are reduced; then we would apply rules to the top of the resulting tree before going on to reduce the third argument further (if needed). We have found that such concurrent rewriting strategies can yield significant savings of space without reducing the amount of useful concurrency.

More generally, it appears that OBJ with E-strategies can be used to specify quite general concurrent processes, for example, protocols. If so, this should be an important advance in specification technology.

2.2.5 Tree Representation

An important question is how to represent trees in the RRM. A basic choice is between *dags* (directed acyclic graphs), which permit sharing identical subtrees, and strict tree structures. The following summarizes our discussions on the relative merits of these representations for concurrent tree rewriting:

1. We first note that the dag representation may consume much less space, since identical subtrees need not be duplicated. The extreme case is to maintain a *fully shared dag*, having no duplicated subtrees at all. However, this is not practical, since it would be very expensive to maintain full sharing during rewriting.

2. Detecting a match may involve determining equality of subterms. For example, to see whether the rule $x + x = x$ applies to a structure, we must check equality of the two substructures matching the variable x. This favors the dag representation, since checking equality may reduce to checking actual identity. However, this yields only a partial gain, because we will not have fully shared dags.

3. Rules with duplicated variables on their righthand side will require copying in the tree representation, but can be implemented just by sharing in the dag representation. Although such rules (called "non-right linear") are quite common, cases that actually require copying large subtrees are rather rare.

4. Sometimes a rewrite can be efficiently implemented just by modifying the content of a cell. This is correct for the tree representation, but for dags it can be wrong. Instead, the part which might have been modified will have to be created independently.

5. For dags, two non-overlapping matches may request information from the same cell in different contexts, thus creating a communication problem that does not arise for trees in the same situation.

6. Both dags and trees require freeing unreferenced items, but dags involve the further issue of controlling multiple replacements of data to avoid damaging shared parts; reference counts or some similar mechanism will be needed.

Quite possibly the debate between these alternatives will be solved in the end by hardware considerations.

3 Ultra High Level Languages

This section describes our work on UHLLs for programming the RRM, beginning with OBJ2 [1]. All of these languages are *declarative*, in the sense that their programs try to state what the problem is, rather than how it should be solved. This allows the compiler and operating system much more leeway in detecting and utilizing whatever concurrency is present in the problem (to the extent that this is possible with available resources). By contrast, today's most popular languages (including parallel languages) are *imperative*, in the sense that their programs consist of commands that tell just what to do and when to do it.

3.1 OBJ2

OBJ2 [1] is a functional programming language with an operational semantics given by tree rewriting, and a mathematical (or "denotational") semantics given by equational logic. Thus, OBJ2 is a declarative language, since its statements have a declarative reading as equations stating properties that one desires the solution to have; in effect, they describe the *problem* to be solved. Moreover, OBJ2 is particularly suitable for the RRM because its operational semantics is tree rewriting. In addition, OBJ2 has a very expressive and flexible type system, including overloading and subtypes; OBJ2 also has user-definable abstract data types (with user-definable "mixfix" syntax) and perhaps the most powerful generic module mechanism available in any current programming language. Moreover, OBJ2 is a "wide spectrum" language that elegantly integrates coding, specification, design, and verification into a single framework.

3.1.1 A Simple Example

The following simple program illustrates some basic features of OBJ2:

```
obj BITS is
  protecting INT .
  sorts Bit Bits
  ops 0 1 : -> Bit .
  op nil : -> Bits .
  op _._ : Bit Bits -> Bits .
  op length_ : Bits -> Int .
  var B : Bit
  var S : Bits
  eq : length nil = 0 .
  eq : length B . S = inc length S .
endo
```

OBJ2's basic entity is the *object*, a module encapsulating executable code. The keywords `obj ... endo` delimit the text of the object. Immediately after the initial keyword `obj` comes the object name, in this case `BITS`; then comes a declaration indicating that the built-in object `NAT` is imported. This is followed by declarations of new data sorts, in this case `Bit` and `Bits`, and of the constants `0 1 nil`, and the operations `_._` and `length_`, each with information about the distribution and sorts of arguments and the sort of the result; underbar characters "`_`" are used to indicate argument places for mixfix operators; thus `_._` is infix and `length_` is prefix. Finally, variables of sorts `Bit` and `Bits` are declared and two equations constituting the body of the object are given; `inc` is the increment (i.e., add 1) function on integers. Trees (i.e., terms) are built up from the constants and function symbols. For example, two such terms are `3 . (7 . nil)` and `length 1 . (8 . (3 . nil))`; the latter evaluates to 3, by applying the two equations as left-to-right rewrite rules.

3.1.2 Modules

OBJ2 has three basic kinds of entity:

- *Objects* declare new types (with operations and sorts of data), and give equations that define the operations. These equations are executable code when interpreted as rewrite rules.
- *Theories* define the requirements of interfaces; they also declare sorts, operations, and equations, but these equations are not executable as rewrite rules.
- *Views* express bindings of actuals to requirements at module interfaces; they are used to put modules together to form larger program units.

The first two together constitute the class of *modules*. Modules can also import other modules, thus making use of the capabilities that they provide. This leads to the OBJ *module hierarchy*.

An important aspect of OBJ2 is its systematic use of generic (parameterized) modules. Encapsulating related code makes it more reusable, and generics are even more reusable, since they can be "tuned" for a variety of applications by choosing different parameter values; moreover, debugging, maintenance, readability and portability are all enhanced. The interface declarations of OBJ2 generics are not purely syntactic, like Ada's; instead, they may contain *semantic* requirements that actual modules must satisfy before they can be meaningfully substituted. This can prevent many subtle bugs. An unusual feature of OBJ2 is the commands that it provides for modifying and combining program modules; thus, (a form of) program transformation is provided within the language itself. A key principle here is the systematic use of *module expressions* for describing and creating complex combinations of modules. This process of putting generic modules together to form larger program units, in which previously written code is described by theories, and also is manipulated by module expressions to produce new code, is called *parameterized programming*; it provides a level above that of conventional programming languages [7].

3.1.3 Progress on OBJ2

The following enhancements of the OBJ2 system have been accomplished:

- A new pattern matching algorithm for combinations of axioms such associativity, commutativity, identity, etc. has been implemented. This algorithm is extensible, so that algorithms for new cases can be easily added.
- Save-restore facilities have been implemented for both OBJ2 text and internal database states.
- The user interface has been substantially enhanced.
- The library of OBJ2 programs has been greatly increased, and several substantial examples (including some programming language interpreters) have been used for debugging.

In addition, we have developed the mathematical and operational semantics of the language, based on order-sorted equational logic; two long papers on these topics are in an advanced stage of completion.

Work is underway to make the evaluation of module expressions (one of the most sophisticated features of the language) and the rule generation process (to generate and optimize the internal form of the rewrite rules) more robust, and to further test the interpreter with some large-sized benchmarks. First release of an OBJ2 interpreter is expected in August 1986.

3.1.4 Towards OBJ3

Our experience with OBJ2 has led to new design ideas that will make OBJ3, a future version of OBJ, substantially more expressive, robust, and efficient. We have been using OBJ2 as a design language for OBJ3. The ideas include:

- A new, simpler and more efficient, operational semantics.
- A more general parameterization mechanism.
- A better interactive parser.
- A better error recovery and error specification mechanism.

OBJ3 design is well advanced, and completion is scheduled for mid-1987.

3.2 Multiparadigm Languages

The RRM is a perfect match for a rewrite-rule based functional language like OBJ. However, our recent work has shown that logic programming and object-oriented programming can be viewed as extensions of OBJ (we have already published two versions of a paper on "Eqlog" [2], combining OBJ with logic programming, and have an advanced draft of a paper on "FOOPS" [3], extending OBJ with object-oriented facilities). Moreover, these extensions are naturally and efficiently supported by the RRM. Support for object-oriented programming only requires implementing more general access to subtrees that store attributes of objects; tree rewriting is still the computational model. Logic programming is a more challenging extension, since it requires the architecture to handle unification as well as tree rewriting. However, tree rewriting and unification are very closely connected, and it is possible to efficiently implement unification by tree rewriting; in this regard, the ideas of Berkling [5] seem very relevant. In summary, the RRM seems an ideal match for UHLLs that combine the most promising programming language paradigms, including functional, logic, and object-oriented programming. Of these, functional programming is the most fundamental (and consequently the one receiving the most current attention) with the others being regarded as extensions. However, we expect the final UHLL for the RRM to support all three major paradigms in an integrated manner.

4 Simulation of Concurrent Tree Rewriting

We have run a number of RRM simulations at the level of concurrent tree rewriting, by modifying and instrumenting our current sequential OBJ2 interpreter. The purpose of this exercise was to determine:

1. How easy is it to program the RRM with OBJ2?
2. How effectively does OBJ2 capture the concurrency that exists in typical problems?

We discovered that it was not difficult to write highly abstract OBJ2 programs that effectively captured the concurrency that existed in the problems considered.

4.1 The Examples

Our earliest simulation studies used some simple standard benchmarks, the Fibonacci numbers, matrix multiplication, and sorting; the results for these examples are presented in some detail, because the examples are familiar and facilitate comparison with other computational models. Many other examples have also been done, including generating the prime numbers in a "lazy stream" and an interpreter for a higher-order functional language. These examples cover a wide range of basic problems: for example, FFT (Fast Fourier Transform) can be viewed as matrix multiplication, and results for matrix multiplication transfer to FFT, and in fact, give essentially performance; also, communication routing problems can be viewed as sorting problems; and the functional language interpreter is typical of many software development tools.

We first wrote very straightforward OBJ2 programs for the Fibonacci, matrix multiplication and sorting problems. For the Fibonacci numbers, we used the usual recursive definition; for matrix multiplication we used lists to represent matrices (thus, a matrix is a list of rows which are lists of matrix elements); and for sorting, we used a simple merge sort.

4.2 The Results

We first investigated the claim that straightforward OBJ programs often exploit much of the concurrency available in a problem. For this purpose, the time required by our current

sequential OBJ2 interpreter was compared with the (virtual) time required to compute the answer with concurrent tree rewriting. The results were as follows:

Problem	Sequential	Concurrent	Ratio
Fibo	a^n	n	exponential
mm	n^3	n	n^2
sort	$n \log n$	n	$\log n$

where a is the golden mean, about 1.618. In each case, the concurrent tree rewriting simulation agrees with the theoretical lower bound for the given algorithm. These results strongly support our claim that natural and simple OBJ programs often exploit much of the concurrency that is available in a problem.

As might be expected, if a Fibonacci algorithm that runs in logarithmic sequential time is coded in OBJ, its execution by concurrent rewriting is only slightly faster than its sequential execution. (It is worth recalling that some problems are *inherently sequential* in the sense that there is a sequential solution which no concurrent solution can outperform.)

We also investigated how much concurrency could be exploited by OBJ. The matrix multiplication and sorting problems were rewritten to use tree structured data rather than lists; this naturally gives rise to "divide-and-conquer" solutions. The resulting programs were only about fifty-percent longer than the straightforward list programs. However, especially for sorting, they were conceptually more difficult; the sorting program was essentially a bitonic merge sort. The results were as follows, where Max Tree Size refers to the concurrent case:

Problem	Sequential	Concurrent	Ratio	Max Tree Size
mm	n^3	$\log n$	exponential	n^3
sort	$n(\log n)^2$	$(\log n)^2$	n	n^2

Comparing these results with theoretical AT^2 lower bounds for VLSI circuits [6], where circuit area A is the maximum tree size arising in concurrent simulation, gives the following:

Problem	Concurrent	AT^2 Lower Bound
mm	$n^3(\log n)^2$	n^4
sort	$n^2(\log n)^4$	n^2

Sorting achieves performance very close to the theoretical lower bound (this is actually typical for realistic VLSI sorting designs). Matrix multiplication shows a simulated performance that is not realizable in practice; i.e., no actual VLSI implementation could attain this performance. This is because the concurrent tree rewriting model does not count the cost of rearranging and duplicating the elements of the matrices involved, which would require significant area in a VLSI circuit.

The maximum tree size for the recursive Fibonacci number program grew very rapidly with n under (simulated) concurrent rewriting. We also found other examples of exponential or hyperexponential growth. Since real machines will have limited tree storage, this is very undesirable. We found that relatively simple programming style guidelines and generalized E-strategies can often reduce the problem to acceptable proportions. For example, it is better to use two conditional equations than a single `if_then_else_fi`, provided that it is easy to evaluate the conditions. Similarly, tuning the iterative Fibonacci algorithm by "specializing" its rules (by replacing a single rule by two or more other more specialized rules whose lefthand sides together cover the original lefthand side) yielded roughly a five-fold reduction in maximum tree size (by forcing bottom-up evaluation). Such transformations are straightforward, and could even be done mechanically.

The concurrent rewriting simulator was constructed by modifying the existing OBJ2 interpreter so that a single simulated concurrent rewriting step is found by:

1. labelling the tree with all possible rewrites,

2. choosing a non-overlapping set of rewrites (by top-down search), and

3. performing all these rewrites.

Statistics were accumulated as the simulation proceeded. This was relatively easy to implement, given the facilities provided by the current OBJ2 implementation.

4.3 Discussion

A major finding is that it is often easy and natural to write algorithms in OBJ2 that exploit most of the concurrency available in a given problem, and thus run surprisingly fast on the RRM. (For the closely related discussion of *scalability*, see Section 5.3.)

Although we found that it was not difficult to get essentially time optimal performance from OBJ programs, we also noticed that some of these programs consumed great amounts of space. However, relatively simple programming style guidelines and generalized E-strategies often reduced this problem to acceptable proportions. E-strategies can be useful when calculations are performed that turn out to be useless, by imposing restrictions on the order in which subtrees are reduced.

5 Architectural Options and Test Facility

From an architectural viewpoint, the RRM provides a unique opportunity to execute ultra high level programs with an unprecedented degree of concurrency. In contrast to most current investigations of advanced computing structures, the RRM project is a symbiosis of software and hardware concerns. We therefore expect to avoid the sad outcome of a "concurrent computer architecture in search of suitable problems" or its dual, a "modern programming paradigm woefully slow on traditional von Neumann computers." Our group is excited by the prospect of a long-awaited breakthrough.

5.1 Architectural Alternatives

We have identified three levels of architectural concern, each with specific problems:

1. The *cell level* is concerned with implementating *rewrite cells*, which store the individual tokens that constitute trees.

2. The *node level* is concerned with organizing a single RRM processor, composed of many cells plus a common controller.

3. The *network level* deals with several RRM nodes cooperating to solve a problem.

The architectural options under consideration for each of these levels are outlined below.

5.1.1 Cell Architecture

A cell stores the tokens that constitute trees and performs simple operations on them. We expect to partition problems so that only a few bits (say 8 to 10) are needed for these tokens. The operations performed on a stored token are:

- Match the token against an externally supplied pattern.

- Cooperate with immediate neighbor cells to determine the success or failure of a pattern match.

- Replace the token with a new value when the pattern match is successful.

Although we are far from a final cell design, it seems clear that it would be desirable to pack as many cells as possible on a single custom VLSI chip. This implies that we should minimize the amount of logic local to each cell. Currently, we favor having all cells in a chip share a common controller, which will broadcast matching and replacement commands. Thus, cells will operate in lock-step (SIMD) mode.

The RRM is not intended to deliver optimal performance for massively numerical computations. If relatively few numerical operations are required, they can be performed on a backup microprocessor (possibly the controller) associated with a number of cells, without loss of efficiency. This is because numerical calculations can occur only at the outermost frontier of a tree, and therefore cannot delay other computations. At a cost of approximately 25% more transistors, numerical calculations could be handled at the cell level, though not in the fastest possible way.

5.1.2 Node Architecture

An RRM node consists of many cells and their controller; it is in effect a block of *active memory*. Assuming that all nodes are implemented on a single VLSI chip to enhance speed, the main issues at this level are:

- The *connection structure* supporting exchange of information among cells.
- The *node controller* and its dialog with cells.

It seems clear that the cells inside a node must be arranged in a regular pattern so as not to waste silicon area. We are still debating how to use this physical structure to support the underlying tree structure that is being rewritten. The two major alternatives provide different performance trade-offs:

- Mapping trees onto the physical interconnection pattern provides very fast intercell communication, but structure changes (resulting from tree rewriting) are very costly.
- Non-local communication channels incur a high performance penalty and may reduce the degree of concurrency, but structure rearrangements become trivial.

We will use extensive simulation to determine the best trade-off between these options.

The node controller broadcasts commands to all cells inside its node. We expect rules to be already compiled into a sequence of simple operations. Due to limited storage capabilities, only active rules will be kept inside a node. This raises issues similar to those associated with "working sets" in von Neumann virtual memory architectures. A further difficulty is the need to avoid decisions based on cell status, since otherwise the controller will become a bottleneck as it serializes cell replies. Much more work is needed in both defining the architecture at this level and in devising efficient compilation techniques.

5.1.3 Network Architecture

This architectural level is concerned with issues of coordinating many rewrite nodes, including

- node communication needs,
- node interconnection structures, and
- rule distribution methods.

RRM nodes are relatively independent entities. Since the resources available at each node are quite limited, nodes must cooperate to solve non-trivial problems. The underlying data tree is a very dynamic object, and may expand or shrink at any moment. The active rewrite region may thus span several nodes. The partitioned concurrent rewriting computation

model (see Section 2) will be used to study the problem of finding "articulation points" in trees where activity should be continued to another node.

Two nodes containing adjacent tokens must cooperate in the pattern matching process. We assume that nodes are each on a VLSI chip, so that internode communication is very slow compared with operations inside of nodes. There are also synchronization issues, since the nodes are independently controlled. We are still evaluating solutions to this problem. The major options are *duplicated memory* and *message passing*:

- The duplicated memory model of node communication allows a few cells in each of the two nodes to contain duplicated information, so that pattern matching is not hindered by node borders. This comes at the expense of more complex node coordination to avoid inconsistency.
- The message passing model relies on nodes sending part of their cell information in a message to another node. This requires more complex pattern matching operations.

Again, the debate between the two models will be resolved by simulation.

Two other important issues are *load balancing* and *network topology*. A node that needs to send part of its activity to another node must find a free node that can be accessed cheaply, or else place some of its tree on backup memory. We need a network topology that will support cheap access to free nodes. The correct solution probably depends on the number of nodes. For a relatively few nodes, a crossbar switch seems practical. For a larger number of nodes, we should try to avoid a fully general routing facility, since it will either be slow, or else very expensive or even impossible. To take advantage of the locality of interconnection that is characteristic of tree-structured data, we are considering a variety of interconnection structures, including a hexagonally tesselated sphere (or torus) and a hypercube.

Rule sets are compiled and distributed from a central minicomputer. Since we envision a large number of RRM nodes, each with a different active rule set, the architecture must avoid the minicomputer bottleneck. One solution currently being explored is the *inheritance of rule sets*. Preliminary simulation results show that when a node spawns another active node, its rule set is often relevant to the new node. Therefore the rule memory can be copied from the old node, instead of waiting for the minicomputer; only rules that are specific to the new node need be requested from the central minicomputer.

5.2 Architectural Testbed

Our discussion of architectural designs has stressed the need for extensive simulation, since we are faced with an enormous decision space, in which functionality, structure, and coordination should all be varied. This motivates some requirements on a simulation facility for the RRM project:

- Modularity is needed to facilitate experimentating with small model changes.
- Our multiple levels of decision need multiple levels of representation and simulation.
- Model development requires separating tasks into
  1. module functionality,
  2. module interconnection structure, and
  3. control and coordination of module activity.
- Easy to use graphics interface.

We will carefully evaluate existing simulators, such as SARA, Palladio, and N.2, but it seems likely that none will satisfy our requirements. Therefore, part of our effort should be devoted to building a custom simulator that can cope with our rich decision space.

5.3 Architecture Scalability and Fault Tolerance

The proposed RRM organization combines two concurrent architectural paradigms:

- Cells within a node share a single controller, and therefore operate in SIMD mode.
- Nodes are independent MIMD processors that cooperate by exchanging messages.

The combination of SIMD and MIMD allows both the speed advantage of cells executing rewrites in lock-step, as well as the flexibility of independently controlled processors.

A particular advantage stemming from independent execution at the node level is *scalability*. In the proposed RRM architecture, whenever existing resources are exhausted by a particular rewrite activity, some nodes may have to stop and wait for other activities to finish. Requests to spawn new nodes will not be honored, but instead must be kept for later processing[3]. If the node interconnection structure is a regular tesselation (e.g., a sphere or torus), more nodes can be added just by expanding the surface, and these additional nodes could take care of pending node generation requests. Thus, a computation that can use more resources will show performance proportional to the number of available nodes. Of course, the node interconnection structure should allow unlimited addition of new node communication channels, but this does not seem especially difficult given the inherent local connectivity of tree rewriting.

Another important issue is *fault tolerance*. In common with most MIMD architectures, the RRM can tolerate faulty nodes, since the node connection structure provides more than sufficient connectivity. More relevantly, the rewrite model of computation can survive failed nodes simply by re-requesting action from other healthy nodes. This feature, common to functional programming languages can guarantee that no partial computation will be lost if some node fails to deliver results.

5.4 Estimated Performance

Although it is early to evaluate the RRM architecture, some educated guesswork is possible, based on a number of assumptions. RRM design called for each rewrite node to be contained on a single custom VLSI chip. Several rewrite nodes can be placed on a single printed circuit board, sharing some backup memory and access to a minicomputer storing the total rule set. Since we plan initial experimental chip designs using DARPA's MOSIS facilities, several constraints are more or less clear:

- Power consumption demands CMOS technology; previous MOSIS experience shows that the largest standard chip can accomodate up to about 250,000 transistors.
- A reasonable guess at the cell complexity yields about 200 logic gates, corresponding to about 1,500 transistors for each RRM cell.
- Packing 128 cells in each chip would leave enough space for the node controller[4].
- One could place 128 nodes (custom VLSI chips) on a single board, together with their interconnections, some backup memory, and a minicomputer interface.
- A reasonable demonstration machine could contain 16 such boards.

This discussion sets a target of 128 rewrite cells per node and 16,384 cells per RRM board. The proposed prototype will then contain 16 boards for a total of about 256 thousand cells. For estimating speed, let us assume that:

- The custom chips operate at 20 Mhz, since there are no complex or lengthy operations involved in tree rewriting.

[3]This could be done by providing message backup memory at the node level.
[4]The choice of cell communication method could change these estimates.

- A rewrite, consisting of a pattern match and a subtree replacement, takes an average of 5 clock cycles.
- The result is a cell having a computational power of 4 MIPS (million instructions per second).
- The total prototype computational power is then (by straight multiplication of the two factors) one million MIPS!

Of course, this result is utterly misleading[5]. First of all, only a fraction of the pattern matches will succeed, so most cells will not actually do replacements. Secondly, internode communication will slow down the computation whenever a pattern is matched across a chip border. Thirdly, the basic instruction is considered to be a single replacement, which is hard to compare with the ultra high level instructions of OBJ programs.

However, the performance estimate of one million MIPS is still valuable. If unsuccessful matches and communication problems permit only .75% of matches to yield replacements[6], the RRM prototype would still have a performance of 8,000 MIPS. Although more simulations are needed to refine these figures, it seems clear that the amount of concurrency available in the RRM promises a tremendous breakthrough in performance.

References

[1] Joseph Goguen, Jean-Pierre Jouannaud, Kokichi Futatsugi and Jose Meseguer. Principles of OBJ2. In *Proceedings, Symposium on Principles of Programming Languages*, page 52, 1985.

[2] Joseph Goguen and Jose Meseguer. Eqlog: equality, types, and generic modules for logic programming. In *Functional and Logic Programming*, page 295, Prentice-Hall, 1986.

[3] Joseph Goguen and Jose Meseguer. Object-oriented programming as reflective equational programming. In preparation, SRI International, 1986.

[4] Gerard Huet and Jean-Jacques Levy. *Computations in Non-ambiguous Linear Term Rewriting Systems.* Technical Report 359, INRIA Laboria, 1979.

[5] Klaus Berkling. *Epsilon-Reduction: Another View of Unification.* Technical Report, Syracuse University, 1986.

[6] Jeffrey Ullman. *Computational Aspects of VLSI.* Computer Science Press, 1983.

[7] Joseph Goguen, Claude Kirchner, Sany Leinwand, Jose Meseguer and Timothy Winkler. *Progress Report on the Rewrite Rule Machine.* IEEE Technical Committee on Computer Architecture Newsletter, to appear 1986.

[5] However, it is no more misleading than the usual computer manufacturers' use of MIPS performance estimates.

[6] This is a very pessimistic assumption - it means that on the average only one cell within each node performs a replacement.


DATE
ILMED
-8